(2)

Ada 9X Project Report

Character Set Issues for Ada 9X

October 1989

Office of the Under Secretary of Defense for Acquisition

Washington, D.C. 20301

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1989	Final Report

4. TITLE AND SUBTITLE

Ada 9X Project Report, Character Set Issues for Ada 9X, October 1989

5. FUNDING NUMBERS

C = MDA-903-87D-00

6. AUTHOR(S)

Ronald F. Brender
John B. Goodenough, editor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

SEI-89-SR-17

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
1211 South Fern St., 3E113
The Pentagon
Washington, DC 20301-3080

Ada 9X Project Office
AF Armament Lab/FXG
Eglin AFB, Florida 32542-5434

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

This report has been produced under the sponsorship of the Ada 9X Project Office. It is one in a series that addresses special issues relevant to the Ada revision effort.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Issues and design considerations relevant to the introduction of large and/or multiple coded character sets in the Ada language definition are presented. Emphasis is on identifying and understanding design and implementation consideration.

14. SUBJECT TERMS

Ada 9X, ANSI/MIL-STD-1815A, Ada Joint Program Office, Ada 9X Project Office

15. NUMBER OF PAGES

50

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED		UL	

NSN 7540-01-280-5500

Standard Form 298.
Prescribed by ANSI Std. Z39
298-01

2

Ada 9X Project Report

Character Set Issues for Ada 9X

October 1989

Office of the Under Secretary of Defense for Acquisition

Washington, D.C. 20301

SEI-89-SR-17
October 1989

Character Set Issues for Ada 9X

Ronald F. Brender

Digital Equipment Corporation

This report has been produced under the sponsorship of the Ada 9X Project Office. It is one in a series that addresses special issues relevant to the Ada revision effort. John B. Goodenough, of the Software Engineering Institute, has served as the editor and coordinator for each report.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This work is sponsored by the U.S. Department of Defense. The views and conclusions contained in this document are solely those of the author(s) and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

Table of Contents

List of Figures

1. Character Set Issues for Ada 9X

Abstract: Issues and design considerations relevant to the introduction of large and/or multiple coded character sets in the Ada language definition are presented. Emphasis is on identifying and understanding design and implementation considerations. Some recommendations are made.

During balloting on the ISO draft international standard for Ada (ISO DIS 8652) conducted in 1986, two comments (neither part of a negative vote) indicated that Ada should be adapted to allow use of coded character sets other than ASCII. Both Japan and Czechoslovakia urged that Ada be adapted to allow the use of so-called "national replacement character sets" as part of the next revision of Ada. ISO JTC1/SC22/WG9 agreed that this proposal "should be considered in the next revision." The draft standard was approved without modification, and became known as ISO 8652-1987 Programming Language Ada.[1]

Now that the revision process for the ANSI, and thereby the ISO, standards is in progress under the leadership of the US Department of Defense, it is appropriate to explore the issues involved in adapting the Ada language definition to deal with coded character sets other than ASCII.

This report is organized in two main parts. Section 2 provides a brief overview of the world of coded character sets. This material provides background for the following main part. Section 3 provides a survey of the various design issues that must be considered for the Ada language. Where possible, approaches that are promising as a basis for incorporation in Ada 9X are recommended.

1.1. Representation, Not Linguistics

This report does not deal at all with any number of linguistic issues that arise in the creation of multilingual or international applications. Such issues include time, date, and monetary formatting, capitalization rules, comparison, and sorting, and on and on. This report focuses solely on underlying text representational issues. Linguistic and representational issues are sometimes confused because one generally can not address solutions for linguistic problems without solving representational problems as well. A premise of this report is that first one can and should solve the

[1] This standard is the same as the US ANSI standard, and perhaps better known by its designation in the United States: ANSI/MIL-STD-1815A-1983.

problems of representing text from around the world and only then work on linguistic issues in the context of a known representational approach.

1.2. Some Terminology

Within the character set standards community, the term "character set" generally means a set of characters independent of any particular encoding, while "coded character set" means a character set together with a specified encoding of those characters. Within the programming language standards community, practitioners are not used to carefully distinguishing between a character set and a coded character set; in particular, the term "character set" is often used when "coded character set" would be more accurate. The unfortunate consequence is that the term "character set" is at best ambiguous and at worst can be a source of confusion and miscommunication between practitioners from the two communities. In an effort to avoid confusion, this report avoids the term "character set" altogether. The terms used here are "character repertoire" for a set of characters independent of encoding and "coded character set" for a character repertoire together with an encoding. (Even this use of the word "repertoire" is not completely comfortable to coded character set experts, who use it only in a very specific fashion in the context of a particular standard — ISO 6937. However, the compatible though slightly broader usage found here should not cause any confusion.)

2. Coded Character Set Standards

The world is awash with coded character set standards. There are international standards, national standards, and a vast array of corporate, institutional, and other private standards. Often, multiple overlapping coded character sets exist for different application domains (newspaper publishing, bibliographic services such as libraries, and on and on) even within a single country. An excellent overview of many important international and national standards can be found in [Clews 1988].

This report will deal almost completely with (some of) the international standards for coded character sets together with a small number of national coded character sets, especially for Asian countries.

2.1. 7-Bit Coded Character Sets

2.1.1. ISO 646 and ASCII

Perhaps the best known coded character set is ASCII, which is the US variant of ISO 646. ISO 646 is a seven bit coded character set, with two main variations.

The first variation is the "International Reference Version" (IRV). The IRV is currently the same as ASCII except that the international currency symbol (¤) is used instead of the dollar sign ($). However, because the dollar sign does dominate in actual practice, this variation is being revised to (return to) using dollar sign.

The second variation is not so much a particular coded character set, but rather a framework for allowing national variations that are closely related to the IRV. In particular, ten characters are considered freely replaceable by other (local) characters,[2] while two characters allow a choice of two alternatives.[3] So-called National Replacement Character (NRC) sets, that is, coded character sets defined according to ISO 646, have been defined for most if not all of the countries of

[2] The replaceable characters are: commercial at (@), left square bracket ([), reverse solidus (\ (back slash)), right square bracket (]), circumflex accent (^), grave accent (`), left curly bracket ({), vertical line (|), right curly bracket (}), and tilde/overline (~).

[3] Pound sign (British currency, £) can replace number sign (#) and/or international currency sign (¤) can replace dollar sign ($).

Western Europe as well as many other parts of the world. (Over 30 such national sets are listed in [Clews 1988].)

While there is some advantage to a family of closely related coded character sets, there are obvious disadvantages as well. The most obvious is that because text is rarely accompanied by identification of the coded character set used, a text composed in one country will generally display or print improperly in another. A second disadvantage is that there are not enough replaceable characters for some countries even in Western Europe (a long standing sore point for the French, for one example).

2.1.2. Other 7-Bit Coded Character Sets

In addition to the NRC 7-bit coded character sets related to ISO 646, there are a wide variety of other 7-bit coded character sets. Many of these are intended to support the languages of a particular geographic area, while others are intended to serve particular application domains or subject matters.

2.1.3. ISO 2022: Combinations of Coded Character Sets

ISO 2022 is the ISO standard that provides a framework for combining and switching among coded character sets. We will briefly sketch the major characteristics of ISO 2022 that are important for the following material. See Figure 2-1.

An 8-bit coded character set is viewed as a combination of two 7-bit coded character sets, where the 8th (high-order) bit distinguishes the two sets. If the high bit is 0, then the GL ("graphics left") set is used, while if the high bit is 1, then the GR ("graphics right") set is used.[4]

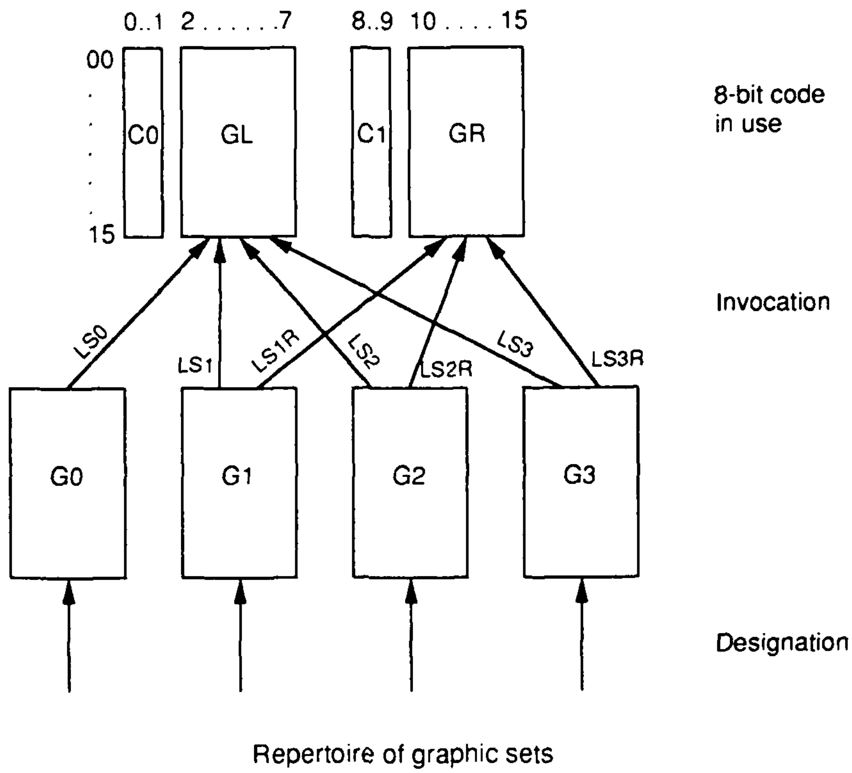
The term "graphics" is in contrast with "controls" — since ISO 646, more modern coded character sets are generally concerned only with graphics, while the possible controls (return, tab, and so on) are the domain of a separate standard (ISO 6429). The "left" versus "right" distinction reflects the conventional manner of presenting a pair of 7-bit code tables. Following the division first established in ISO 646, the first 32 codes of each 7-bit code set are reserved for controls. These are known as the C0 and C1 control areas, respectively.[5]

The two 7-bit coded character sets currently "active" are but two of four sets, known as G-sets (G0, G1, G2, G3) that have been designated. Escape sequences of three, four, or possibly more bytes are used to name an arbitrary coded character set from a registry operated under the terms of ISO 2375 and administered by ECMA (the European Computer Manufacturers Association). Any particular device may not support the designated set, in which case the effect of the designation is implementation-defined.

Once a coded character set has been designated into a G-set, that G-set can then be invoked

[4]"GL" and "GR" are informal names given to the groups of codes from 32 to 127 and 160 to 255, respectively. These names are not used/defined in the relevant ISO standards; they are introduced here because they are convenient names to have available.

[5]The C1 controls thus include the codes from 128 to 159, inclusive.



Notes:

1. LS*n* and LS*n*R lock the G-set G*n* into GL or GR, respectively
2. SS2 and SS3 (single shift) shift the G-set G2 or G3, respectively, into GL for one character

Figure 2-1: *ISO 2022 Code Structure*

into either GL or GR (except that G0 can only be invoked into GL). One or two byte controls are defined for performing these invocations and there are both locking and non-locking (single shift) forms. The invocation controls can be thought of as a short alias for an arbitrary coded character set, so long as that coded character set has been designated into a G-set. The G-sets and which ones are invoked thus constitute important state necessary to properly interpret incoming characters.

An analogy may be useful in understanding these ideas. Consider the the 256 combined codes of C0, GL, C1 and GR taken together to be a virtual address space, where the contents (meaning) of each GL or GR "address" is a glyph to be displayed. Invoking a G*n* set into GL or GR is like mapping a physical block of memory into the active virtual address space, thereby changing the meaning (contents) of the virtual addresses. Designating a coded character set is like reading new contents into one of the G-sets (whether from an I/O device or other physical memory).

There are further aspects of ISO 2022 that will be introduced later.

2.2. 8-Bit Coded Character Sets

In recent years, emphasis has shifted from 7-bit to 8-bit coded character sets, and from loosely coordinated variations to more closely coordinated variations. The ISO 8859 family of coded character sets is the primary example of this, but it will be helpful to first look briefly at ISO 6937 and even revisit ISO 646.

Within ISO 646, characters that required accents were generally formed as a sequence consisting of the base character, a backspace, and the accent (the order of the base character and accent can be exchanged as well). This scheme has several disadvantages:

- While suitable for hardcopy devices that can "overstrike," the scheme is much less well suited for CRT-based display devices.
- The use of varying length representations for characters is awkward for software.
- Look ahead is generally needed to determine whether a complete logical character has been found.
- The ambiguity in the order of the base character and its accent adds further complexity.

ISO 6937 attempted to ameliorate some of these difficulties by defining a coded character set in which the accents were non-spacing; that is, did not advance the imaging position. The following base character completed the character. (Free standing accents can be imaged as the accent followed by space.) This reduced the size of accented characters from three to two bytes and eliminated ambiguity and look ahead problems; however, varying character size and imaging problems remained. This code found significant application in European telematic services, but very limited support among major computer vendors.[6]

ISO 8859, formulated in multiple parts, defines a series of coded character sets, which are summarized as follows:

Part	Informal Name	Geographic Area Served
1	Latin-1	Western Europe
2	Latin-2	Eastern Europe
3	Latin-3	Southern Europe
4	Latin-4	Northern Europe
5	Latin/Cyrillic	
6	Latin/Arabic	
7	Latin/Greek	
8	Latin/Hebrew	
9	Latin-5	Western Europe (variation)

Each part defines an independent, complete 8-bit coded character set in its own right. However, these sets have a great deal in common. Most importantly, all of them define the first 128 codes (the equivalent of GL in the ISO 2022 description given earlier) as equivalent to the new ISO 646 International Reference Version (IRV), that is, as equivalent to ASCII. In addition, where the

[6]This description of ISO 6937 intentionally omits discussion of the registration of subsets (repertoires) as not important to the purposes of this report.

same non-ASCII character occurs in more than one part, that character is coded with the same value as much as possible. Finally, the groupings are designed to match regions that are economically closely related, as well as geographically.

The main technical characteristic of importance here is that all characters in ISO 8859 are coded as a single 8-bit byte — accented characters are considered distinct characters and are coded separately from their related non-accented characters.

2.3. 16-Bit Coded Character Sets

2.3.1. ISO 2022-Based Representations

In addition to 7-bit sets, ISO 2022 also allows a designated coded character set to consist of two, three or even four (7-bit) bytes. When this alternative is used, it is required that all of the bytes of a character have the same high-order bit setting (all zero or all one).

In the Asian countries, the most common application of this capability is illustrated in Figure 2-2.

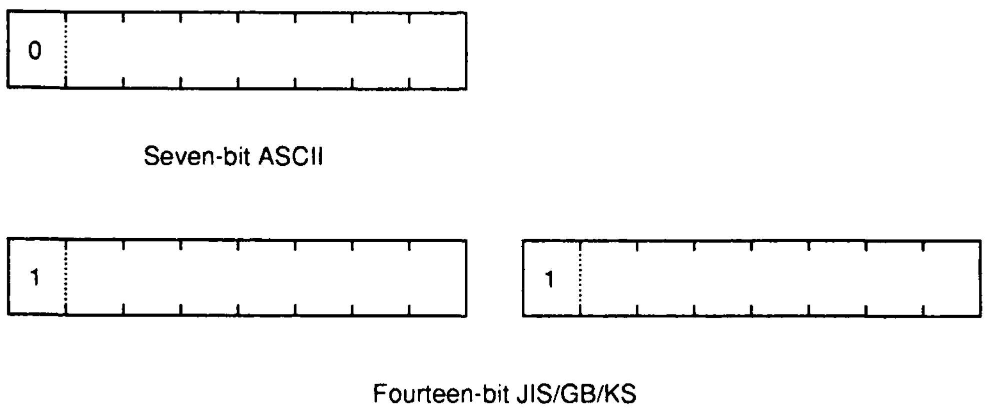


Figure 2-2: Mixed One-Byte/Two-Byte Representation

Typically, ASCII (or a closely related NRC set) is designated and invoked into GL and a local two-byte set is designated and invoked into GR. The resulting combination can be thought of as a variable length coded character set, but this is really a bit of a misnomer.

The two-byte character set is more properly thought of as a 14-bit set, rather than a 16-bit set. In particular, the same character set can just as well be invoked into GL as into GR; if invoked into GL, characters would then be represented with both high order bits set to zero. The point is that, under ISO 2022, the high order bit of each 8-bit byte is really control information that is part of the overall coding structure and not part of any particular coded character set.

A second reason that it is not appropriate to consider the combination as a coded character set proper, is because the two component sets have many characters in common. Good coded character set design dictates that a "proper" character set does not code the same character in more than one way. Each of the major two-byte sets (China, Japan, and Korea) includes a complete set of both upper case letters duplicating those of ASCII (but no accented characters).

An alternative view of the coding space used by this mixed one-byte/two-byte representation can be formed by looking at a two-dimensional code plane where the first byte is one axis and the second byte is the other axis. In order to include the one-byte characters in this two-dimensional space, we will assume that that byte is prefixed by some fixed value,[7] thereby obtaining a uniform two-byte virtual code. See Figure 2-3.

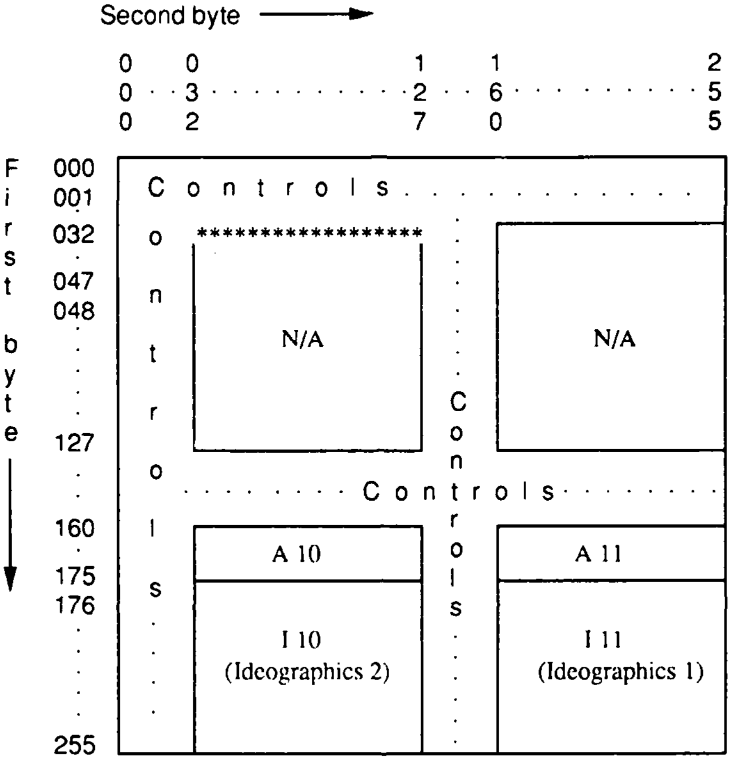


Figure 2-3: Mixed One-Byte/Two-Byte Code Plane

In this view, the one-byte characters consist of just a half row of characters in the row with the assumed first byte value of 32. This half row is shown using asterisks (*) in Figure 2-3. The requirement that both bytes of a two-byte code have the same high-order bit value means that two-byte codes are limited to the lower-right quarter of the plane. That is, characters are limited to the area labeled A 11 and I 11 in Figure 2-3.

There is no real technical advantage served by the ISO 2022 requirement that all characters have the same high order bit setting. In some systems, this requirement is dropped with the result that a second quarter, the lower-left, of the code plane can be used. This area is labeled A 10 and I 10 in Figure 2-3.

Note that in any case, the entire upper half of the code plane, with the exception of the single half row, is not available in this coding scheme.

[7]The choice is arbitrary; we will use 32 because it is the first non-control code and for other reasons that will become clear later.

This mixed length representation has the "nice" property that if all of the characters of a string or record consist of just the ASCII characters, then the string or record has the form of a simple ASCII string or record. This is frequently exploited to achieve a measure of upward compatibility with systems of American or European origin.

Many systems in Asia restrict support to just two statically determined coded character sets: ASCII in the GL area and one of the Chinese, Japanese or Korean sets (as appropriate to the system location) in the GR set. The so-called Extended Unix Code adds statically determined G2 and G3 sets, which are conventionally used only by means of the SS2 or SS3 single shifts (never the locking shifts). In Japan, the JIS X0201 one-byte Kana code is assumed designated into G2, with G3 left available for user definition. In other areas, both G2 and G3 are available for user definition.

Typical software is not designed to handle designation of other coded character sets into any of the G-sets. Note, in particular, that in the absence of the support for coded character set designation, simultaneous support for Chinese, Japanese and Korean is not possible.

The effect of these conventions is very much analogous to the national replacement set situation, wherein the GL/G0 characters are common but the GR/G1 (as well as the G2 and G3) character codes get interpreted differently from system to system.

2.3.2. Shift Code Based Representations

ISO 2022 is not the only basis for handling large coded character sets, of course. In the mainframe world, one very common scheme consists of combining two separate code sets, one a one-byte code that includes roughly the same characters as ASCII and the other a two-byte code for the local script. A one-byte "shift out" control code changes the mode from one- to two-bytes, while a one-byte "shift in" control changes the mode back to one-byte characters.[8]

2.3.3. Shift Range Based Representations

A kind of halfway scheme between the high-bit convention and shift control scheme is the so-called "Shift JIS Code" used in Japan, especially on personal computers. In this scheme, bytes with codes 0 to 127 are interpreted as one-byte ASCII, codes 160-191 and 192-223 are interpreted as one-byte Katakana (according to Japanese standard JIS X0201), and codes 128-159 and 224-255 are combined with the following byte to form a two-byte code that is interpreted as a Kanji character from JIS X0208.[9] The two-byte codes of shift JIS encode the same set of characters as JIS X0208, as well as encoding them in the same order as JIS X0208; the individual character codes are not the same, of course.

[8] "Shift Out" and "Shift In" controls are also defined as part of the older 7-bit mode of ISO 2022.

[9] For those familiar with the typical 8-bit code table, codes 128-159 are the C1 controls from the 8/x and 9/x columns, while codes 192-223 are the 12/x and 13/x columns (which are not used by JIS X0201).

2.3.4. A Perspective

While the variety of schemes for combining coded character sets may seem extensive and complicated, there are really but a small number of variations involved. In general, any given code can be considered to have both a graphic effect and a control effect.

The pure shift code schemes, where each code is either a graphic or a control, seem conceptually simplest. But, such schemes are actually quite complicated to use because implicit state or context (what was the most recently encountered shift?) must be known or determined to correctly interpret any given code.

The ISO 2022 scheme, where every code is both a graphic (or part of one) *and* a control is actually cleaner in most ways, at least when limited to two coded character sets, because it avoids the need for implicit state or context.

The shift JIS scheme is a non-standard variation on the ISO 2022 scheme that allows more single byte characters to be represented at the expense of reducing the number of two-byte characters than can be represented.

2.4. Multiple-Octet Coded Character Set

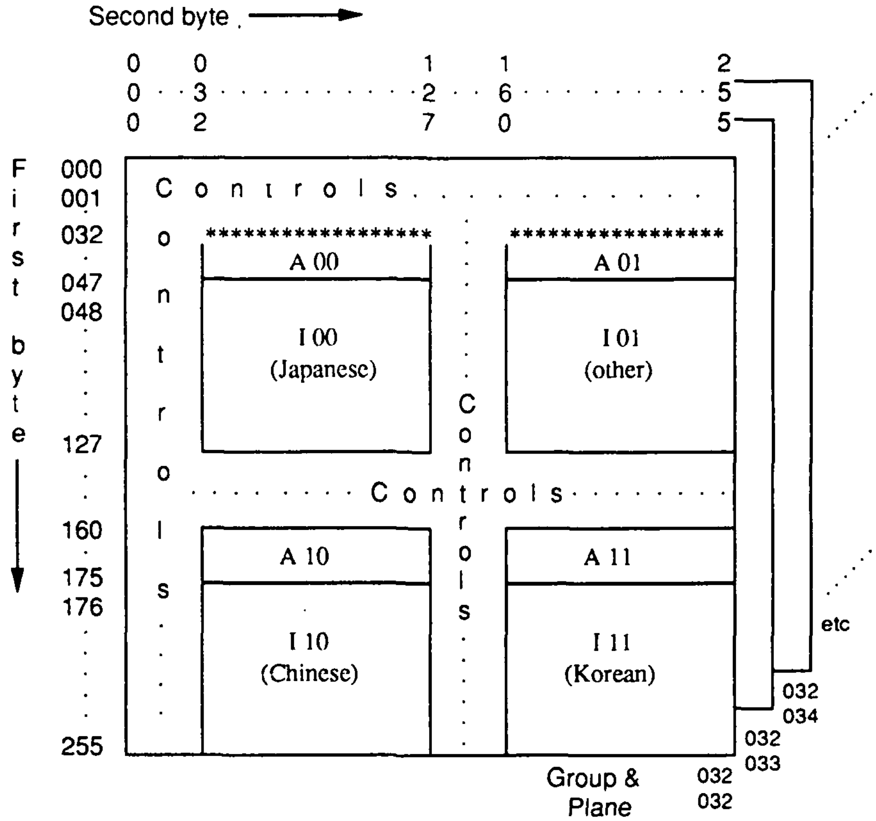
ISO/IEC JTC1/SC2/WG2 is currently developing a worldwide coded character set, called the Multiple-Octet Coded Character Set. ISO DP 10646 was published in early 1989. A second draft proposal (DP) is planned for late 1989. The WG2 committee is working toward publication of a DIS (Draft International Standard) document in early 1990 and the final standard in late 1991 or early 1992. Only a few characteristics of this code are important to this presentation and these are discussed below. See Figure 2-4.

The complete code is considered to consist of four octets.[10] This code is regarded as a single entity, consisting of 65,536 "planes" of characters, divided into 256 "groups" of 256 planes each. Only the first group is being standardized at this time, leaving extensive room for expansion. Each plane consists of 256 "rows," with each row containing 256 "cells."

For compatibility with existing 8-bit facilities, character codes wherein any octet has a value in the range 0 to 31 or 127 to 159 is *not used*. This allows the common C0, DEL (delete), and C1 control codes and related sequences to be used in combination with the multiple-octet code without needing to know that it is a multiple-octet code. It follows that graphic characters are restricted to four quadrants within any given plane.

The first usable plane, plane 32 of group 32, is termed the multilingual plane. It generally includes the characters of all of the alphabetic scripts of the world (including Eastern and Western European, Greek, Cyrillic, Arabic, Hebrew, Maldivian, Syriac, Indian, etc.), as well as up to 7680 characters (ideograms) each for Japanese Kanji, Chinese Hanzi, and South Korean Hangul and

[10] An "octet" is an eight-bit byte. The term octet is used because the term byte is sometimes used with various numbers of bits. This standard defines a coded character set specifically in an eight-bit environment.



Multilingual plane is Group 032, Plane 032

A xx = 16*96 character alphabetic zone
I xx = 80*96 character ideographic zone
Q xx = a quadrant (A xx + I xx)

ISO Latin-1 is contained in Group 32, Plane 032, Row 032
Example: Latin (uppercase) A has code 032 032 032 065

Figure 2-4: Multilingual Plane from ISO DP 10646

Hanja. In addition, up to 7680 "restricted use"[11] characters are included that are used for alternative presentation forms[11] (such as ligatures), or for dynamically redefinable (DRCS) or private use characters.

A one-octet subset, row 32, of the multilingual plane is (by design) identical to the ISO Latin-1 character set.

---

[11] SC2/WG2 has decided on some changes in terminology for use in the second DP, as follows:

Old (from first DP)	New (for second DP)
Discernibly Different Zone	Restricted Use Zone
Presentation Variant	Presentation Form
Form-of-use	Compaction Method

Several "compaction methods"[11] are envisioned:

- Four octets per character
- Three octets per character
- Two octets per character from the multilingual plane
- One octet per character from the first graphic row of the multilingual plane, which is the same as the ISO Latin-1 character set

All but the first may also be used in variant forms that allow use of a proposed new control character, SGCI (Select Graphic Character Introducer), that precedes a single four-octet character.

The compaction method is intended to be persistent, i.e., not subject to frequent adjustment. For transmission-related contexts, persistence means that the compaction method is negotiated once at the time a connection is established and not changed thereafter. For programming language contexts, persistence means that the compaction method is fixed throughout and for the lifetime of a given string object or text file.

For applications where dynamic compaction is warranted, a fifth compaction method is defined that allows operating temporarily in specified one-, two-, or three-octet subsets (the assumed high-order octets of each character are dynamically specified by control sequences), as well as switching amongst subsets.

While the one-octet Latin-1 and two-octet *multilingual subsets will* be suitable for many purposes, it should also be clear that the full four-octet code will be needed for others, especially (but not only) in the Asian countries.

Terminology

A note of caution about terminology is in order. In the context of the Asian two-byte standards, the term "plane" is generally used to refer to a 14-bit set of characters. In the context of the ISO MOCS standard, the term "plane" is used to refer to a 16-bit set of characters, of which each 14-bit subset is called a "quadrant."

3. Ada Issues

This section provides a survey of design issues that must be addressed as part of incorporating increased coded character set support in the Ada language.

It is often hard to chose a preferred order of topic presentation because of the many interactions of different issues — the reader's patience is assumed. Following is a summary of the design issues discussed in this section.

- Character data types:
    - Choice of stream versus typed vector model
    - Binding of predefined data type(s) to specific coded character sets
    - Number of predefined character types
        - Implementation-defined
        - Based on the ISO MOCS (draft proposed) standard
    - Lexicographic ordering
    - Type conversion
    - Effect on TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO
- Source program independence of underlying coded character set representation
- Allowed usage of extended characters
    - In identifiers
    - In literals and the effect on 'IMAGE and 'VALUE
        - Universal character and string types with implicit conversion
        - Overloaded on all character or string types (in scope)
    - In reserved words
    - In implementation-defined pragmas and attributes
    - Bi-directional text
- Input/output representation
    - File representation versus processing representation
        - Names for coded character sets and coding structures
        - File attributes
    - Character counts versus byte counts versus display positions
    - Input methods

3.1. Character Data Type(s)

The following sections address a number of related issues having to do with the nature and number of character data types.

3.1.1. Streams Versus Vectors of Typed Components

There are generally two models for defining an encoding of strings in high level languages: a stream model and a vector of typed components model. In the stream model, the character represented by a given sequence of bits depends on the interpretation given to preceding characters. For example, if one character signals a shift to two-byte codes, then subsequent bytes are interpreted two at a time, until a character specifies a shift to a different coding. Alternatively, a code may indicate a shift to a different coded character set, in which case, a following string of bits may represent a different character than it did before the shift.

In the vector model, every character occupies the same number of bits and the meaning of a given bit representation is the same wherever it appears in the string. Ada, of course, models strings as vectors.

When reviewing these alternative ways of representing strings, representatives at an ISO SC22[12] Ad Hoc Meeting on Character Handling Requirements in Programming Languages concluded that neither could be uniformly recommended over the other; rather, programming languages may provide functionality from either or both models as appropriate to their intended application domains [SC22 Ad Hoc 1989-B].

> *The stream model deals directly and explicitly with the real representation in the form defined by relevant coded character set standards. Its advantages are flexibility and efficiency for some kinds of operations. However, applications exploiting these characteristics may be difficult to construct and may not be portable to other representations (because details of escape sequences or other coded character set characteristics tend to be incorporated into the application code). The stream model complicates some traditional string processing operations (indexing, substringing, concatenation).*
>
> *The typed vector model views character information as a sequence of same-sized character objects. This facilitates many string processing operations.*
>
> *Support of more than one coded character set in the typed vector model requires both conversions (usually at the I/O boundary) and sufficient size of character objects for the domain of characters to be processed. Its advantages are simplicity of use (details of coded character set standards, escape sequences, etc. are hidden) and portability (execution is with a consistent character representation with conversion to "real" representations at the I/O boundary). Disadvantages are efficiency and space (conversion and manipulation of larger objects).*

---

[12]SC22 is the subcommittee of ISO/IEC JTC1 concerned with programming languages. SC2 is the subcommittee of ISO/IEC JTC1 concerned with coded character sets.

Representatives of the several working groups of SC22 (including this author as the representative of WG9, Ada) met in March 1989 to discuss common character set issues and to prepare recommendations and requests to SC2 based on programming language requirements. Some of the same SC22 representatives (again including this author) then met with representatives of SC2 (as well as SC21 concerning data bases) in April 1989 to present and discuss those recommendations and requests. [SC2/SC22 Ad Hoc 1989-A and -B] present the results of those meetings.

The space/time trade-offs that lead to distinguishing integer vs. floating point, and even several kinds of floating point, are seen to also apply to large character sets. The choice of model is then dependent on the intended use of a programming language and, as is the case with numerics, no single model can be recommended.

Ada, of course, is clearly in the group of languages that model a string as an array (vector) of characters. Moreover, Ada's language-defined abilities to index, to substring, and to rename components and substrings preclude any practical extension of the existing model to include stream-like characteristics. For example, Ada allows components of strings to be renamed. Such a capability is incompatible with the stream model since an assignment to an earlier component of the same string might change the size of the renamed component or shift the memory location of the component (e.g., when a large component is replaced by a smaller one), or change the character set associated with the component.

If a stream-oriented model were desired, it must be introduced through new language features rather than as a modification and/or reinterpretation of the existing model. The balance of this report does not consider the stream alternative further.

Recommendation 1: *Continue the "array of typed components" approach to strings as already established in Ada. Do not pursue any stream-oriented alternative.*

3.1.2. Binding Character Types to Coded Character Sets

Ada currently defines its single character type in a manner that is equivalent to ASCII. While an implementation might manage to use a different encoding than ASCII during execution, the need to preserve the user semantics of the ASCII encoding for type CHARACTER makes this approach unlikely.[13] For example, if an array is indexed by type CHARACTER, the A^{th} element of the array must be the element following CHARACTER'PRED('A') and preceding CHARACTER'SUCC('A'). String comparison operations similarly must use the ASCII collating sequence.

A key Ada language strategy question is whether to maintain this intimate association with the ASCII character set. Recall that Ada is, in fact, unique among the standardized programming languages in having such a close association. FORTRAN, COBOL, LISP, and so on, have all been effective vehicles for portable programs even though they limit their requirements to listing a set of characters that must be representable while avoiding any association with any particular coded character set.

On the other hand, changing the definition of type CHARACTER to be coded character set independent does not, of itself, do much to address how to deal with large and/or multiple character sets in Ada. It seems unlikely that type CHARACTER could be so flexibly defined that an implementation could use, say, the ISO MOCS (four-octet) character set without affecting the semantics of existing programs in subtle ways. For example, the value of CHARACTER'LAST would become implementation-dependent, so the number of loop iterations dependent on this value

[13]This author is not aware of any implementation that uses a run-time character set other than ASCII as the encoding for type CHARACTER.

would change. In addition, selection of the **others** choice in a case statement over values of type CHARACTER would become implementation-dependent. Finally, the implicit dependence on the size of CHARACTERs as 7- or 8-bit entities, both internally in programs and externally in data files, probably could not be changed without unacceptable re-engineering cost. In short, any change in the number of literals of the predefined type CHARACTER would probably require some kind of modification to most Ada programs that do any kind of non-trivial text manipulation. As a rough guess, for example, this would include a significant number of the programs currently held in the Ada repository.

If extending type CHARACTER from 7-bits to 8-bits were sufficient to deal with the variety of character set needs worldwide, then the resulting incompatibilities might be considered warranted. However, given that a single character type (even associated with a coded character set other than ASCII) is not viable, it is necessary to consider approaches involving multiple character types.

Recommendation 2: Do not change the predefined types CHARACTER and STRING.

3.1.3. The Number of Builtin Character Types

With so many coded character sets of interest in different parts of the world, it seems impossible to both serve the inherent needs reflected by these sets as well as maintain a language design that employs a reasonably small number of character types. Two approaches are of interest, however.

3.1.3.1. Multiple Implementation-Dependent Sets

One approach is to allow each implementation to introduce any number of additional character data types, with each type corresponding to a coded character set of local interest. These data types might be named according to some scheme, say, CHARACTER_*n*, where *n* is a number.

The main disadvantage of this approach is that any use of such extended character types would seriously compromise the portability of the program. Without some means for selecting the right character type based on the coded character set supported (for example, a character type able to represent Greek), a program would easily become non-portable between two implementations that support the same coded character set but happen to use different type names for those sets.

A variation might be to specify a repertoire of character set names, each with a specific associated coded character set. The repertoire must be extensible over time, so some kind of registration mechanism would be needed. Possibly the naming scheme could be based on the codes that are already defined for the escape sequences used to designate character sets in the ISO 2022 model, in which case the registration process administered by ECMA in accordance with ISO 2375 would automatically serve Ada as well.

While this general approach might be built into a viable model, it would do little to simplify the already considerable problems of operating in an exceedingly rich coded character set world.

3.1.3.2. ISO MOCS Based Character Sets

A second approach is to define three additional character types based on the ISO MOCS standard as follows:

LATIN1_CHARACTER	One-octet subset (same as ISO Latin-1)
MULTI_CHARACTER	Two-octet subset (multiiingual plane)
MOCS_CHARACTER	Full code

This gives a sequence of increasingly general character types, where CHARACTER is a subset of LATIN1_CHARACTER, which is a subset of MULTI_CHARACTER, which is a subset of MOCS_CHARACTER. Further, ISO SC2 intends to maintain the MOCS code as *the* universal character set. Thus, this approach will serve the needs of any geographic area as rapidly as the needed script is incorporated in the ISO standard. Note that the initial multilingual plane will already achieve this for a very large fraction of the world, whether measured in terms of population, geographic area or economic activity.

Complementing these character types would be corresponding string types (LATIN1_STRING, MULTI_STRING, and MOCS_STRING) having the above character types as their respective component types.

The main disadvantage of this approach is that the two-octet subset will be required for some regions where existing one-octet sets are already in use (for example, Israel, Greece, the Arabic countries, and others). Only Western Europe retains the "advantage" of a one-byte code. Of course, Ada does not provide any kind of capability for these regions at the moment. Thus this disadvantage is only relative to some other hypothetical proposal that might allow for multiple 8-bit character types.

Recommendation 3: Add the LATIN1_CHARACTER, MULTI_CHARACTER, MOCS_CHARACTER, LATIN1_STRING, MULTI_STRING, and MOCS_STRING types as described above in the predefined package STANDARD. Require these types to be present in all implementations. (It is a separate matter whether an implementation allows source text to include characters other than those of ASCII in source programs [see below].)

Compatibility Issues: Adding a new data type to package STANDARD as proposed above introduces a small potential incompatibility with existing Ada programs, namely, if these identifiers are declared in other packages in existing programs, then a use clause will no longer make them visible. For example:

```
package CHAR_SETS is
    type MOCS_CHARACTER is (...);
    type MOCS_STRING is array (POSITIVE range <>) of MOCS_CHARACTER;
end CHAR_SETS;

with CHAR_SETS; use CHAR_SETS;
package P is
    A : MOCS_CHARACTER;          -- 1
    ....
end P;
```

Given the existing definition of package STANDARD, the variable A is of type CHAR_SETS.MOCS_CHARACTER. If Recommendation 3 is followed, the variable A will be of type

STANDARD.MOCS_CHARACTER, because as RM 8.4(6) notes, a directly visible declaration cannot be hidden by the effect of a use clause. Since STANDARD.MOCS_CHARACTER is directly visible, CHAR_SETS.MOCS_CHARACTER is not made directly visible, so the program has a different meaning. Such programs are unlikely to exist in practice, and even so, any use of these new identifiers can be easily discovered with a preprocessor.

Another minor incompatibility is that expressions such as

```
"ABC" = "DEF" and 'C' = 'D'
```

will be illegal, because there is no way to resolve the types of the string literals and character literals, or the equality operators. (This issue is discussed further in Section 3.3.3.) Expressions comparing the values of two string literals or two character literals are, however, unlikely to be written by human programmers; they are more likely to appear only in machine-generated code, and such generators can usually be modified to cope with this difference.

3.1.4. Lexicographic Ordering

Ada defines the ordering operators for arrays as follows [RM 4.5.2(9)]:

> *The ordering operators <, <=, >, and >= that are defined for discrete array types correspond to* lexicographic *order using the predefined order relation of the component type. A null array is lexicographically less than any array having at least one component. In the case of nonnull arrays, the left operand is lexicographically less than the right operand if the first component of the left operand is less than that of the right; otherwise the left operand is lexicographically less than the right operand only if their first components are equal and the tail of the left operand is lexicographically less than that of the right (the tail consists of the remaining components beyond the first and can be null).*

Although this definition is satisfactory for the predefined CHARACTER type, it is unlikely to be satisfactory for any of the new character types. For example, in the Latin-1 character set, it would be inconsistent with standard practice to consider the accented characters to be greater than 'z' simply because they appear later in the enumeration type declaration. Consequently, any programs using the extended string types will undoubtedly want to redefine the ordering operators for these types. However, the subject of sorting and collating sequences is a big and complicated problem in its own right. The Ada community should look to SC22 to develop a common framework that can apply across programming languages.

3.1.5. Type CHARACTER as an Explicit Enumeration Type

Appendix C of the Ada standard defines the type CHARACTER by means of an explicit enumeration declaration. This is unlikely to be feasible for future multiple, large character sets, no matter how they are formulated. However, this is believed to be more of a presentation problem than a problem of any real semantic significance.

3.1.6. Type Conversion

The ability to use the names of numeric types to perform type conversions should be extended to the character and string types. If a character to be converted is not representable in the target type, then an exception should be raised.

For the character types specifically, type conversions can be made available without other language enhancement by ensuring each of the types is related by derivation (see Figure 3-1). In this Figure, MOCS_CHARACTER is declared as the basic type, and each of the other types is derived using appropriate range constraints. Type conversions are then freely available among all of these types in accordance with RM 4.6(8-9).

```
type MOCS_CHARACTER is
         (. . . first 128 literals as in Appendix C. . . ,
          . . . second 128 as for Latin-1 . . . ,
          . . . literals for the two-octet set. . . ,
          . . . the remaining literals for ISO 2022) ;
for MOCS_CHARACTER'SIZE use 32;

type MULTI_CHARACTER is new MOCS_CHARACTER range
         MOCS_CHARACTER'VAL(0) .. MOCS_CHARACTER'VAL(2**16-1);
for MULTI_CHARACTER'SIZE use 16;

type LATIN1_CHARACTER is new MULTI_CHARACTER range
         MULTI_CHARACTER'VAL(0) .. MULTI_CHARACTER'VAL(255);
for LATIN1_CHARACTER'SIZE use 8;

type CHARACTER is new LATIN1_CHARACTER range
         LATIN1_CHARACTER'VAL(0) .. LATIN1_CHARACTER'VAL(127);
```

Figure 3-1: Declaration of the Extended Character Types in STANDARD

Unfortunately, corresponding type conversions among the proposed string types does not quite come for free. Such type conversions would be available according to LRM 4.6(10-11) for array types except that the several string types do not have the same component type. A minimal generalization to the rules for array types is to allow conversions where the component types are convertible because they are related by derivation. Such a conversion is just a change of representation when converting from a component type with reduced range (such as CHARACTER) to a target string type whose component type has an extended range (such as MOCS_CHARACTER). Of course, the reverse conversions are more costly, because each string component must be checked to be sure it belongs to the range of the target component type. Although such checks would be costly, the ability to convert between at least the predefined string types is so important that it should be provided in one way or another.

Recommendation 4: Provide the ability to perform type conversions among the predefined string types.

3.1.7. The Input/Output Packages

However many predefined character types are defined in Ada and/or allowed to be added as an implementation-defined option, ideally all of them ought to be considered co-equal and interoperable with respect to the input/output packages. For example, the NAME function ought be able to return the file name in any of the predefined string types, for any file object, independent of the character set used to name the file in the OPEN or CREATE operation, and independent of the character set of the data stored in the file.

However, for those subprograms that take more than one parameter of a string type (OPEN and CREATE), it seems extravagant to cater to the cross-product of possible type combinations. It seems sufficient that there should be one subprogram for each distinct string type, e.g., OPEN should accept NAME and FORM parameters that are both of type STRING, LATIN1_STRING, MULTI_STRING, or MOCS_STRING. Should an application need to call OPEN or CREATE using different string types for the NAME and FORM parameters, it should suffice to require that the arguments be converted to a common string type.

Overloading input/output services in this manner will bring to string-oriented callable services the same kind of ambiguity issues that exist for numerically-oriented callable services. For example, suppose there are several subprograms for PUT in package TEXT_IO, one for each of the several character types. Then, a simple call such as PUT('A'); would be ambiguous. For numeric services such as the mathematical packages defined by the Ada Numerics Working Group, the typical approach is to define the service as a generic package that is instantiated by each application as needed. Where qualification is still required, selected component notation can be used to resolve ambiguities.

In order to create a generic TEXT_IO package, it will be necessary to add a "generic formal character type" to the language. A generic formal character type is needed to allow strings to be written within the generic TEXT_IO package, e.g., the default expressions for the FORM and NAME parameters of the CREATE and OPEN operations. It would also ensure that text-oriented generic units are only instantiated with a character type (rather than with any discrete type).

<u>Recommendation 5</u>: *Introduce a generic version of TEXT_IO named, say, GENERIC_TEXT_IO with two generic formal character type parameters named, say, NAME_CHARACTER and DATA_CHARACTER (see Figure 3-2). Default generic formal string parameters named, say, NAME_STRING and DATA_STRING are defined using these component types, respectively. Use the formal type NAME_STRING in place of type STRING in the CREATE, OPEN, NAME and FORM procedures, and use the formal type DATA_STRING in place of STRING in the GET, GET_LINE, PUT and PUT_LINE procedures. Use the DATA_CHARACTER type in place of CHARACTER in the PUT and GET procedures.*

Similar generic packages should be provided for SEQUENTIAL_IO and DIRECT_IO, to provide uniform capabilities for the OPEN and CREATE procedures.

With this formulation, the existing predefined package TEXT_IO will then be provided as an instantiation:

```
with GENERIC_TEXT_IO;
pragma ELABORATE (GENERIC_TEXT_IO);
package TEXT_IO is new GENERIC_TEXT_IO(
    CHARACTER, CHARACTER,
    STRING,    STRING);
```

In a similar manner, provide additional predefined packages as follows:

```
with GENERIC_TEXT_IO;
pragma ELABORATE (GENERIC_TEXT_IO);
package LATIN1_TEXT_IO is new GENERIC_TEXT_IO(
    LATIN1_CHARACTER, LATIN1_CHARACTER,
    LATIN1_STRING,    LATIN1_STRING);
```

```
with IO_EXCEPTIONS;
generic
    type NAME_CHARACTER is ('');      -- a character type is required
    type DATA_CHARACTER is ('');      -- a character type is required
    type NAME_STRING is array (POSITIVE range <>) of NAME_CHARACTER;
    type DATA_STRING is array (POSITIVE range <>) of DATA_CHARACTER;
package GENERIC_TEXT_IO is

    ...

    procedure CREATE (FILE : in out FILE_TYPE;
                      MODE : in FILE_MODE       := OUT_FILE;
                      NAME : in NAME_STRING     := "";
                      FORM : in NAME_STRING     := "");

    ...

    function  NAME    (FILE : in FILE_TYPE) return NAME_STRING;

    ...

    procedure GET     (FILE : in FILE_TYPE; ITEM : out DATA_CHARACTER);
    procedure GET     (ITEM : out DATA_CHARACTER);

    ...

    procedure PUT     (FILE : in FILE_TYPE; ITEM : in DATA_STRING);
    procedure PUT     (ITEM : in DATA_STRING);

    ...
end GENERIC_TEXT_IO;
```

Figure 3-2: Sketch of generic TEXT_IO package

The notation ('') is introduced to signify a formal generic character type, i.e., the corresponding actual parameter must be a character type.

```
with GENERIC_TEXT_IO;
pragma ELABORATE (GENERIC_TEXT_IO);
package MULTI_TEXT_IO is new GENERIC_TEXT_IO(
    MULTI_CHARACTER, MULTI_CHARACTER,
    MULTI_STRING,    MULTI_STRING);

with GENERIC_TEXT_IO;
pragma ELABORATE (GENERIC_TEXT_IO);
package MOCS_TEXT_IO is new GENERIC_TEXT_IO(
    MOCS_CHARACTER, MOCS_CHARACTER,
    MOCS_STRING,    MOCS_STRING);
```

Note that the potential for different name and data types is not exploited for the predefined character and string types in the above instantiations. For example, the NAME and FORM parameters for LATIN1_TEXT_IO.OPEN are both of type LATIN1_STRING, and so is the input parameter for PUT_LINE. However, a different instantiation could create a PUT_LINE with a different string type, while OPEN and CREATE require parameters of type LATIN1_STRING.

Of course, such a generic package can be instantiated with user-defined character and string types such as ROMAN_CHARACTER and ROMAN_STRING:

```
type ROMAN_CHARACTER is ('I', 'V', 'X', 'L', 'D', 'M');
type ROMAN_STRING is array (POSITIVE range <>) of ROMAN_CHARACTER;
```

Instantiating the generic TEXT_IO package with these types for NAME_CHARACTER and NAME_STRING would undoubtedly lead to peculiar effects. If the generic approach is adopted, probably the specification should state that the effect of the instantiation is not defined by the language if the instantiation is performed with any types other than the predefined character and string types.

3.2. Source Representation

Section 2 highlighted the variety of representation schemes currently in use; moreover, as the newer character set standards, most particularly the ISO MOCS standard, come into use and the needs of non-Western Europe begin to be addressed within programming language and other standards, it is likely that the variety will actually increase, at least for a while.

In the midst of such diversity, it is important for the portability of (Ada) programs that the meaning of a program not be dependent on the underlying coded character set used to represent that program. Historically, there has been a tendency to formulate programming language extensions for Asia, in particular, in ways that reflect and incorporate the current mixed one-byte/two-byte representations into the language design. It is strongly recommended that this tendency be avoided.

The following model has been recommended to other programming language groups [COBOL 1988] [SC22 Ad Hoc 1989-A]. It is repeated here for Ada as well:

> *A character may be represented physically in one or more forms. The different forms of a character, whether associated with different character sets, whether associated with different compression mechanisms, or whether physically of the same size or not, are considered to be logically equivalent and representing the same character. If different character sets are used, then the means for distinguishing (including shifting) between them is implementation-defined. (Any shift characters or other such control information embedded in the source program, if used, are relevant only to identifying the characters making up the source program and are not themselves among the (logical) characters of the source program.)*

Traditional lexical and syntactic specifications are then expressed in terms of the resulting logical character set. It remains an implementation matter whether the conceptual mapping from physical to logical character set occurs as a distinct prepass or whether it occurs only implicitly and on-the-fly.

Note that this recommendation does not preclude, of itself, implementation-defined extensions that do reflect properties of an implementation's particular local representation conventions. An example might be a pseudo-string constant, perhaps expressed using an attribute notation, that yields a value using mixed size characters. However, it helps make clear that such constructs are not encouraged by the programming language standard and are not likely to be portable in any useful manner.

This recommendation is in many ways a compile-time version of the run-time recommendations for separation of processing and file representations during input/output. See Section 3.4, Input/Output, for further discussion.

Recommendation 6: Adopt the statement on source program representation given above.

3.3. Extended Character Usage in Source Programs

The whole purpose of extending the Ada string and character types is to allow source programs to use these extended codes in character literals and string literals. Given such usage, it is natural to consider the use of extended characters in program identifiers as well. Both topics are considered in this section.

For the purposes of this section, it is convenient to have a term for characters other than characters defined in ASCII that are suitable for use in identifiers, that is, excluding special characters such as punctuation or (free standing) accents.[14] We will call these "extended name characters."

3.3.1. Referring to Extended Characters in Literals

If the source program is written using characters outside the ASCII set, say, in Latin-1 characters, it is a simple matter to write string literals and character literals using these characters:

```
E_ACCENT : LATIN1_CHARACTER := 'é';
SLOGAN   : constant STRING := "Liberté, Egalité, Fraternité";
```

However, suppose the source program representation has no way to express such characters, e.g., suppose only the ASCII character set is supported. It would be rather inconvenient and error-prone to write:

```
E_ACCENT : constant LATIN1_CHARACTER := LATIN1_CHARACTER'VAL(2#1110_1001#);
SLOGAN   : constant STRING := "Libert" & E_ACCENT & ...;
```

Ada currently provides standard names for otherwise unrepresentable ASCII characters. While it might be possible to continue this approach for Latin-1, the approach breaks down completely for the two-octet and four-octet sets. Perhaps alternate notations should be explored for writing literals that fall outside the set that can be used in the source code representation supported by a particular compiler.

3.3.2. Extended Name Characters in Identifiers

It is desired and desirable that identifiers used in a program should be able to include characters from any script representable in the underlying source representation. The SC22 Ad Hoc meeting in March 1989 made a recommendation to SC22 that it direct its working groups to proceed to allow this capability in each of their respective standardized languages. (SC22 will consider this recommendation at its plenary meeting in September 1989.) It would be preferable, however, if

[14]This definition conveys the general idea but definitely needs refinement. The goal is to generally allow any locally meaningful nouns or phrases to be used as or in identifiers. (No requirement that identifiers must be linguistically meaningful is intended, of course.)

there were a common set of identifier characters allowed across standardized programming languages.

Recommendation 7: *Allow extended characters in source program identifiers. Look to SC22 to develop a common set of such characters for use across programming languages, if possible.*

3.3.3. Overloading Resolution Issues

Given the possibility of extended characters in identifiers, the question of overloading resolution for character and string literals bears revisiting. For concreteness, the following example illustrates some interesting issues:

```
type ENUM is (ENGLISH, FRANCAISE, KANJI);
OE : ENUM := ENUM'LAST;
OB : BOOLEAN;
...
B := ENUM'IMAGE(OE) = "KANJI";
```

where, let us suppose, KANJI (two occurrences, including inside the string literal) is in fact the Japanese word for Kanji as written using the Japanese script.

As Ada is defined today, the IMAGE attribute returns a value of type STRING. But given that the attribute is supposed to produce a string representing the corresponding enumeration literal, and given that an extended character is used in writing such an enumeration literal, it is clear that the attribute can no longer satisfy its intent if it is limited to returning just a STRING value.

There are several options for resolving this problem:

(a) Redefine the IMAGE attribute to return a value of type MOCS_STRING. Such a return type is sufficient to represent any enumeration literal given using extended characters in a source program. (Similarly, the VALUE attribute would be redefined to take an argument of type MOCS_STRING.)

(b) Overload the IMAGE attribute to return values of type STRING, MULTI_STRING, and MOCS_STRING, and redefine the VALUE attribute to accept arguments of any of these types. The IMAGE attribute will raise CONSTRAINT_ERROR if the enumeration literal cannot be represented in the selected return type.

(c) Redefine the IMAGE attribute to return a type *universal_string*, which could be implicitly converted to any predefined string type.

Solution (a) suffices to allow the representation of any possible enumeration literal given in source code using extended characters. But this solution would not be very satisfactory for current Ada users, since existing programs that use the IMAGE or VALUE attributes would have to convert the result or argument to type MOCS_STRING. This would undoubtedly be unacceptable to most current Ada users.

Solution (b) has a few disadvantages also. The expression ENUM'IMAGE(OE) = "KANJI" will be ambiguous since there is insufficient information to select from among the visible equality operators. Of course, this expression is currently legal,[15] although probably such expressions

[15]IMAGE's return type, STRING, suffices to resolve both the string literal and the equality operator.

occur in few, if any, real application programs. In any event, it is likely that most uses of the IMAGE attribute will occur in contexts that suffice to determine a unique resolution. So although it may seem obvious that ENUM'IMAGE(OE) = "KANJI" should always be legal, it is unclear how much language change is worthwhile to achieve this effect.

The third solution is analogous to the existing solution for attributes like LENGTH — instead of overloading the attribute LENGTH on all integer types in scope, LENGTH is defined to return the type *universal_integer*. Hence an expression such as ARR'LENGTH = 6 is always legal, because it uses the equality operator for type *universal_integer*. If an analogous *universal_string* type were introduced and string literals were considered to be of this type, then ENUM'IMAGE(OE) = "KANJI" would similarly always be legal because "=" would resolve to the *universal_string* equality operator. Given the existence of a *universal_string* type, it would probably also be reasonable to provide the corresponding *universal_character* type, and to introduce the notion of named constants of types *universal_character* and *universal_string*.

Recommendation 8: The concept of universal character and string types is attractive [to this author] but may be considered a more extensive change than necessary. The Design Notes and dialogues leading to the introduction of the universal numeric types should be reviewed as a source for additional insight. In any case, the particular solution chosen is believed to be less critical than some other issues in terms of overall impact and capability.

3.3.4. Extended Name Characters in Reserved Words

There appears to be no interest in allowing extended name characters in language-defined reserved words. That is, not even "local language" aliases seem to be of interest. (It is reported that there was once a French standard for a version of FORTRAN that had French keywords — **allez** for **goto**, and so on. It was never popular and was soon dropped as a failure.)

Recommendation 9: Do not provide or allow local language aliases for language-defined reserved words.

3.3.5. Extended Name Characters in Implementation-Defined Entities

There appears to be little point to allowing extended name characters in implementation-defined names, such as attributes and pragmas, unless those entities provide a function that is specific to a particular language. (No plausible example comes to mind, though someone will surely suggest one.) There seems no way to rigorously state such a "reasonableness" rule regarding the usage of extended name characters in implementation-defined entities, so the suggested strategy is simply to disallow it.

Recommendation 10: Do not allow extended characters in reserved words or in the names of implementation-defined constructs such as pragmas and attributes.

3.3.6. Bidirectional Names

A number of coded character sets support languages in which text is rendered and read from right-to-left, notably Hebrew and Arabic. However, numeric quantities are generally presented and read from left-to-right (as in Latin languages). Thus, the date

May 21

might be rendered (using English letters in the Semitic order) as in

21 yaM

Nonetheless, the proper internal order of the characters for computer processing is that of the first form, because that is the logical order for reading and writing. The transformation in presentation order for purposes of listing, for example, is an issue that should be separate from and outside of the Ada language.

Good quality transformations are, however, complicated and subtle to infer from simple streams of characters. As a result, specialized control functions have been introduced in ISO 6429 to allow text to contain within itself information on how it should be rendered.[16] Some kind of accommodation of the needs of right-to-left scripts and, more generally, mixed right-to-left and left-to-right text needs to be developed. Preferably a common approach should be developed that applies to all ISO standardized languages. However, this author lacks the expertise to make a concrete recommendation.

<u>Recommendation 11</u>: *Provide no language-defined features or constructs for dealing with or supporting right-to-left or bi-directional text. Look to SC22 to develop common strategies suitable for use across programming languages.*

3.4. Input/Output

The following sections address a number of related issues having to do with input/output.

3.4.1. File Representations Versus Processing Representations

In any system that admits of more than one coded character set and/or text representation in simultaneous use, the issue of character set conversion will surely arise in one form or another. In particular if a common text representation in the external environment is a variable sized one and the programming language supports (only) same-sized characters, then some form of conversion must occur between the two — the key question is where and how it should be invoked.

The SC22 Ad Hoc meeting in March considered this issue at some length and concluded that for languages supporting the typed vector model, coded character set and/or representation conversion should occur as part of input/output as needed. For example, a program written to use the full (four-octet) MOCS code internally can read files containing data written using ASCII, mixed sized ASCII and Kanji, or whatever. On output, conversion from internal processing form to a specified file representation applies as well.

There are a number of advantages to this kind of approach:

- The external representation used in files and the internal representation used during processing need not be the same. This provides flexibility that is very desirable as part of any kind of evolutionary approach for introducing support for large character sets.
- External file representations can be chosen to suit the needs of individual localities

[16]These controls are called Select Presentation Direction (SPD), which sets an overall default presentation direction, and Start Reversed String (SRS), which specifies the points of text reversal including nested reversals.

without needing to change applications (assuming those applications employ a suitably general representation internally). This can ameliorate the disadvantages, for example, of not supporting a large number of character data types for multiple one-byte coded character sets.

- This approach complements the use of the ISO MOCS code as the basis for character data types.

The obvious means to specify this information is by means of either the FORM parameter or a new optional parameter on the Ada OPEN and CREATE subprograms. When this information is omitted, then implementation-defined defaults would apply.

3.4.1.1. Names for Coded Character Sets and Coding Structures

To make this work, there needs to be a means to name coded character sets and coded character set representations. Preferably, naming standards of this kind should come from the relevant ISO SC2 committees and be usable across programming languages rather than something designed by and specific to Ada. In this regard, the SC22 Ad Hoc committee made this recommendation as a request to SC2:

> *Programming languages require a portable way to refer to [i.e., to name] coded character sets and coding structure.*
>
> - *a name must be provided for every coded character set and coding structure;*
> - *this name shall be constructed from the [upper case] letters A-Z and digits 0-9;*
> - *these names shall be unique and indicate the edition of the coded character sets and coding structure, for example ISO8859P1R1987.*

SC2 has yet to act on this recommendation. However, SC2 is aware of similar needs arising in the ASN.1 (Abstract Syntax Notation No. 1) domain as defined in the ISO 8824 and 8825 standards; SC2 is cooperating with other ISO groups to develop suitable naming conventions.

3.4.1.2. File Attributes

When an existing file is open, the most valuable default coding structure and coded character set to assume is what was actually used in creating and writing the file. The SC22 Ad Hoc group concluded that:

> *Programming languages need to know the encoding scheme names and coded character set names associated with existing files.*
>
> *File systems should maintain this information as an attribute of each file.*

Since there are as yet no standardized names for coded character sets or file representations, Ada need not attempt to provide language-defined support for these concepts.

Recommendation 12: *In the absence of standardized names for coded character sets and file representations and, in the absence of generally available file attributes that record file representations, provide no language-defined means to specify or query this information. Enhanced implementation-defined capabilities can be provided by means of the existing FORM parameter of CREATE and OPEN procedures and the FORM function.*

3.4.2. Character Count Versus Byte Count Versus Column Position

In Asian countries, it is common practice today for text to be represented in a combination of two (sometimes more than two) character sets. Section 2.3 discusses several such examples and variations.

While the combinations of character sets vary, it is nevertheless typical for the normal output presentation on a display or printer of the characters of the two-byte character set to occupy twice the width of the characters of the one-byte character set. (For convenience, these shall be referred to as wide and narrow presentation widths in the following.)

This disparity in presentation width resulted historically from the greater intricacy and complexity of ideographic characters, which simply require more resolution (pixels) to present legibly than Latin characters. However, there are certain convenient aspects to this coupling of character code size and character presentation width. In particular, it can simplify some output report formatting tasks: the number of bytes may not equal the number of characters but it does equal the number of print positions. Even where this programming advantage does not obtain, there is still something of a human factors advantage. Latin characters are presented in a manner and density typical of usage in Western countries and ideographic characters are presented in a manner and density typical of usage in the Asian countries. (Actually, in Asian typography, ideographic characters have a width more on the order of one and half times the width of Latin characters. This "odd" ratio has been made available only rarely in computer devices.)

Assuming that there is good reason for users and applications to move from the current combination of character codes to the MOCS code, it would appear that users may be forced to give up the narrow versus wide presentation distinction. (In any case it is clear that the coupling between character width and presentation width is gone.)

One might argue that the loss of differing presentation widths is quite acceptable. While there may be some short term awkwardness associated with continued use of old devices, modern technology (for example, bit-mapped displays and laser printers) is rapidly making a common width for Latin and ideographic characters feasible in a way that is sufficiently pleasing for routine use. Further, the simplification from returning to a programming regime where character count is proportional to presentation width is too valuable to give up lightly. Finally, if any complexity is introduced to handle different presentation widths then fully general proportional font issues should be addressed, not just two widths.

Recommendation 13: In the absence of cross-language conventions for dealing with variable sized fonts (whether associated with one or more character sets), interpret all "column" or "position" specifications in the input/output packages as referring to character count (a "logical character position") independent of the coded character set and/or amount of storage used to represent characters, and independent of the presentation of those characters on the output device. Look to SC22 to develop additional capabilities and conventions in a manner suitable for use across programming languages.

3.4.3. Input Methods

The large number of characters in the Asian character sets creates interesting problems concerning how to input (type) text consisting of such characters. The days of experimenting with keyboards with thousands of keys are long gone — today virtually all Asian systems use some kind of software mediated scheme where the user makes multiple key strokes which are then translated into the appropriate character code. Some methods are based on phonetic transliteration, while others are based on the strokes and shapes making up a character. A few even involve direct entry of the character code as a number! Some even adapt to the content of the text being entered and use artificial intelligence and computational linguistics concepts to increase accuracy. There are literally hundreds of such methods and many of them are highly proprietary.

There seems no need for the Ada language to recognize the use of input methods in any manner. It is the output of the input method that constitutes the Ada program.

Recommendation 14: The existence and operation of input methods, if any, should be transparent to an Ada program.

3.5. Conclusion

This report has discussed various issues concerned with extending the Ada language to handle coded character sets other than ASCII. Although there are numerous design issues to be considered, the necessary changes do not appear to be very extensive or to impose much difficulty on implementers. Hence, it appears quite possible to meet requirements of non-U.S. Ada users without having a significantly negative impact on existing Ada code or on Ada users in the United States.

References

[Clews 1988] John Clews. Language Automation Worldwide: The Development of Character Set Standards. (British Library Reports, 5962). SESAME Computer Projects, North Yorkshire, 1988.

[COBOL 1988] CODASYL COBOL Committee Working Paper, DEC-WP88001. "Issues Regarding Multiple-Octet Character Sets." (Ron Brender) 7 Oct 1988.

[SC2/SC22 Ad Hoc 1989-A] Requirements for Characters in Programming Languages: A Report from the Ada Hoc Joint Meeting of SC2, SC22, and SC21/WG3, ISO/IEC JTC1/SC22 N622R. 26-28 April 1989.

[SC2/SC22 Ad Hoc 1989-B] SC22 Requirements for Character Handling in Programming Languages: A Report from the Ada Hoc Joint Meeting of SC2, SC22, and SC21/WG3, ISO/IEC JTC1/SC22 N623R. 26-28 April 1989.

[SC22 Ad Hoc 1989-A] SC22 Ad Hoc Meeting on Character Handling Requirements in Programming Languages, ISO/IEC JTC1/SC22 N611. "Source Representation Independence of Character Sets." (Ron Brender) 24 Feb 1989.

[SC22 Ad Hoc 1989-B] SC22 Ad Hoc Meeting on Character Handling Requirements in Programming Languages, ISO/IEC JTC1/SC22 N623. 8 March 1989.

See also the ISO and national standards listed in Appendix A.

Appendix A: Summary of Some Coded Character Set Standards

A.1. Corresponding ECMA and ISO Coded Character Sets

ECMA-6	ISO 646	7-Bit Coded Character Set
ECMA-35	ISO 2022	Code Extension Techniques
ECMA-43	ISO 4873	8-Bit Coded Character Set Structure and Rules
ECMA-48	ISO 6429	Control Functions for Coded Character Sets
ECMA-94	ISO 8859	8-Bit Single-Byte Coded Character Sets
	Part 1	Latin Alphabet No. 1 (Western Europe)
	Part 2	Latin Alphabet No. 2 (Eastern Europe)
	Part 3	Latin Alphabet No. 3 (Southern Europe)
	Part 4	Latin Alphabet No. 4 (Northern Europe)
ECMA-113	ISO 8859	8-Bit Single-Byte Coded Character Set
	Part 5	Latin/Cyrillic Alphabet
ECMA-114	ISO 8859	8-Bit Single-Byte Coded Character Sets
	Part 6	Latin/Arabic Alphabet
ECMA-118	ISO 8859	8-Bit Single-byte Coded Character Sets
	Part 7	Latin/Greek Alphabet
ECMA-121	ISO 8859	8-Bit Single-Byte Coded Character Sets
	Part 8	Latin/Hebrew Alphabet
ECMA-128	ISO 8859	8-Bit Single-Byte Coded Character Sets
	Part 9	Latin Alphabet No. 5 (Western Europe variation)
n/a	ISO 10646	Multiple-Octet Coded Character Set (DP)

A.2. Asian Coded Character Sets

A.2.1. Chinese (PRC) Coded Character Sets

GB 2312-1980	Chinese Character Coded Character Set for Information Interchange — Basic Set
GB 7589-1987	Chinese Character Coded Character Set for Information Interchange — 2nd Supplementary Set (see note 2)
GB 7590-1987	Chinese Character Coded Character Set for Information Interchange — 4th Supplementary Set (see note 2)

Notes:

1. The above character sets include simplified characters. Additional sets (at least three) for traditional forms are also planned.
2. GB 7589 and 7590 are approved but have not yet been published
3. Each of the above is a 14-bit (two-byte) character set.
4. GB 2312 includes not only the characters of ASCII, but also complete sets of characters for the Greek and Russian scripts, as well as Japanese Kana characters.

A.2.2. Chinese (ROC) Coded Character Sets

CNS 11643-1986	Standard Interchange Code for Generally-Used Chinese Characters
CCCII	Chinese Character Code for Information Interchange
CISCII	Chinese Industry Standard Code for Information Interchange

Notes:

1. The above are 14-bit (two-byte) character sets.
2. The above include traditional characters. Up to sixteen sets are planned.
3. CCCII and CISCII are de facto standards without official sanction; however, both have significant following. CISCII is an early draft of CNS 11643.

A.2.3. Japanese Coded Character Sets

JIS X0201-1976	Code for Information Interchange (JIS-Roman and JIS-Katakana)
JIS X0208-1983	Code of the Japanese Graphic Character Set for Information Interchange

Notes:

1. JIS X0201, formerly known as JIS C6220, defines two 7-bit set character sets. JIS-Roman is the same as the ASCII character set except that the yen sign replaces the reverse solidus (backslash) and overline replaces tilde. (Note that overline is an allowed alternative rendition for tilde in ISO 646.)
2. JIS X0208, formerly known as JIS C6226, is a 14-bit (two-byte) character set. X0208 includes not only the characters of X0201 as well as ASCII, but also complete sets of characters for the Greek and Russian scripts as well. This standard has "Level 1" and "Level 2" subsets (Level 2 is the complete set).
3. A supplementary set, consisting of an additional quadrant, is under development and expected to be published in late 1989 or early 1990.

A.2.4. South Korean Coded Character Sets

C 5601-1987	Korean National Standard Graphic Character Set for Information Interchange

Notes:

1. The above is a 14-bit (two-byte) character set.

A.2.5. Thai Coded Character Sets

TIS 620-1986	Thai Character Code for Computer
KU Code	Kasetsart University Character Set

Notes:

1. KU code is a de facto code which has a significant following.

Appendix B: Summary of Recommendations

Recommendation 1: Continue the "array of typed components" approach to strings as already established in Ada. Do not pursue any stream-oriented alternative.

Recommendation 2: Do not change the predefined types CHARACTER and STRING.

Recommendation 3: Add the LATIN1_CHARACTER, MULTI_CHARACTER, MOCS_CHARACTER, LATIN1_STRING, MULTI_STRING, and MOCS_STRING types to the predefined package STANDARD. Require these types to be present in all implementations.

Recommendation 4: Provide the ability to perform type conversions among the predefined string types.

Recommendation 5: Introduce a generic versions of TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO that allow instantiations with any of the extended set of character and string types.

Recommendation 6: The Standard should not specify how source code is represented.

Recommendation 7: Allow extended characters in source program identifiers. Look to SC22 to develop a common set of such characters for use across programming languages, if possible.

Recommendation 8: The concept of universal character and string types is attractive but may be considered a more extensive change than necessary. The Design Notes and dialogues leading to the introduction of the universal numeric types should be reviewed as a source for additional insight. In any case, the particular solution chosen is believed to be less critical than some other issues in terms of overall impact and capability.

Recommendation 9: Do not provide or allow local language aliases for language-defined reserved words.

Recommendation 10: Do not allow extended characters in reserved words or in the names of implementation-defined constructs such as pragmas and attributes.

Recommendation 11: Provide no language-defined features or constructs for dealing with or supporting right-to-left or bi-directional text. Look to SC22 to develop common strategies suitable for use across programming languages.

Recommendation 12: Provide no language-defined means to determine the encoding of text files.

Recommendation 13: *Interpret all "column" or "position" specifications in the input/output packages as referring to character count (a "logical character position") independent of the coded character set and/or amount of storage used to represent characters, and independent of the presentation of those characters on the output device. Look to SC22 to develop additional capabilities and conventions in a manner suitable for use across programming languages.*

Recommendation 14: *The method used to create the representation of an Ada source program should not have any effect on the meaning of the program.*

Index